SELFLESS INHERITANCE

John C. Lusth

Department of Computer Science, University of Alabama, Tuscaloosa AL, USA
lusth@cs.ua.edu

ABSTRACT

Formal treatments of inheritance are rather scarce and those that do exist are often more suited for analysis of existing systems than as guides to language designers. One problem that adds complexity to previous efforts is the need to pass a reference to the original invoking object throughout the method call tree. In this paper, a novel specification of inheritance semantics is given. The approach dispenses with self-reference, instead using static and dynamic scope to accomplish similar behaviour. The result is a methodology that is simpler than previous specification attempts, easy to understand, and sufficiently expressive. Moreover, an inheritance system based on this approach can be implemented with relatively few lines of code in environment-passing interpreters.

KEYWORDS

Inheritance, delegation, extension, reification, variation, manipulating scopes

1. INTRODUCTION

It seems to be a much harder task to define inheritance than to decide if a programming language provides a sufficient inheritance mechanism or not. This can be seen from papers themed "Inheritance is..." or "Inheritance is not..." [1-4]. Even as late as 1996, inheritance has been described as a "controversial mechanism" [5]. Such a lack of consensus has been the impetus to describe inheritance formally. Formal descriptions range from Touretzky's [6] work on multiple inheritance in AI systems (but with applications to programming languages), to Cardelli's description of the subtyping relation [7], to Cook's [8] or Reddy's [9] denotational semantics approach.

One aspect to Cook's approach, in common with other formal descriptions of the semantics of inheritance, is the prominence of the self-reference. Indeed, Cook goes so far as to say "manipulation of self-reference is an essential feature of inheritance", meaning the passing of a self-reference to the original invoking object (either as a hidden or formal parameter) to all object methods is a requirement. Taivalsaari, in [5], concurs. As will be shown, this requirement is sufficient, but not necessary. Manipulation of static scopes is sufficient for providing an inheritance mechanism that conforms to common intuition and does so without the need of self-references. This "selfless" inheritance greatly simplifies a formal and pragmatic treatment of the behaviour of an inheritance hierarchy. Moreover, unlike a denotational approach, the new approach can be directly and easily implemented in a language processor to provide a reasonable inheritance mechanism. A clue as to the simplicity of selfless inheritance can be found in the use of concatenation to provide inheritance in Javascript [10]. Concatenation inheritance is closely related to selfless inheritance (but still requires the passing of object references). Selfless inheritance simplifies the task of adding inheritance even further.

Any specification of inheritance semantics must be (relatively) consistent with the aforementioned intuition about inheritance. With regards to the pragmatics of inheritance, there seems to be three forms of inheritance that make up this intuition. Taking the names given by Meyer [11], the three are *extension*, *reification*, and *variation*. In extension inheritance, the heir simply adds features in addition to the features of the ancestor; the heir is indistinguishable from the ancestor, modulo the original features. In reification inheritance, the heir completes, at least partially, an incompletely specified ancestor. An example of reification inheritance is the idea of an abstract base class in Java. In variation inheritance, the heir adds no new features but overrides some of the ancestor's features. Unlike extension inheritance, the heir is distinguishable from the ancestor, modulo the original features. The three inheritance types are not mutually exclusive; as a practical matter, all three types of inheritance could be exhibited in a single instance of general inheritance. Any definition of inheritance should capture the intent of these forms.

In the following sections, a description of how environments in an environment-passing interpreter can be used as the foundation of an object system is given. The next section describes how scopes can be manipulated to provide the aforementioned forms of inheritance. The next two sections cover a possible implementation of two forms of single inheritance, followed by a discussion of the semantic difference between selfless inheritance and the form found in Java. The paper then concludes with commentary on the advantages of selfless inheritance.

It should be noted that this paper focuses solely on the runtime *behaviour* of inheritance hierarchies and does not delve into the (mainly static) issues of polymorphism. Moreover, although the backdrop of this paper is environment-passing interpreters, the results presented herein are relevant to other types of languages. This is because objects are implemented as environments and environments are simply records (with links to other like records). However, environment-passing interpreters are especially suited for this treatment since environments and functions for manipulating them often pre-exist; much of the implementation can be readily reused to implement objects with inheritance.

2. SIMPLE ENCAPSULATION

The three hallmarks of object-orientation are encapsulation, inheritance, and polymorphism. Before we tackle inheritance, we first need the ability to encapsulate data, that is, to make simple, general purpose objects.

A notion that simplifies our definition of inheritance is to use environments themselves as objects. Since an environment can be thought of as a table of the variable names currently in scope, along with their values, and an object can be thought of as a table of instance variables and method names, along with their values, the association of these two entities is not unreasonable.

Thus, to create an object, we need only cause a new scope to come into being. A convenient way to do this is to make a function call. The call causes a new environment to be created, in which the arguments to the call are bound to the formal parameters and under which the function body is evaluated. Our function need only return a pointer to the current execution environment to create an object. Under such a scenario, we can view the function definition as a class definition with the formal parameters and local variables serving as instance variables and locally defined functions serving as instance methods. Note that this view differs from Reddy [9] in that the local state and the method environments are considered one and the same. The combination of the two serve as a foundation for removing the need for self-reference. Another difference is that Reddy views closures (function records) more akin to objects.

The Scam language [12], which incorporates earlier ideas [13,14] allows the current execution environment to be referenced and returned. Scam code superficially resembles Scheme and

anyone familiar with Scheme or other Lisp-like languages should readily understand the code fragments presented herein. Here is an example of object creation in Scam:

```
(define (bundle a b)
    (define (total base) (+ base a b))
    (define (toString) (string+ "a:" a ", b:" b))
    this)    ;return the execution environment

(define obj (bundle 3 4))
(inspect ((dot obj display)))  ;call the display method
(inspect ((dot obj total) 1))  ;call the total method
```

The variable *this* is always bound to the current execution environment or scope. Since, in Scam, objects and environments the same, this can be roughly thought of as a self reference to an object. The *dot* function (equivalent to the dot operator in Java) is used to retrieve the values of the given instance variable from the given object. The *inspect* function prints the unevaluated argument followed by its evaluation.

Running the above program yields the following output:

```
((dot obj display)) is "a:3, b:4"
((dot obj total) 1) is 8
```

It can be seen from the code and resulting output that encapsulation via this method produces objects that can be manipulated in an intuitive manner.

It should be stated that encapsulation is considered merely a device for holding related data together; whether the capsule is transparent or not is not considered important for the purposes of this paper. Thus, in the above example, all components are publicly visible. To make encapsulation semi-transparent, keywords such as private can be used to customize encapsulation (with a price of syntactic and implementation complexity) or, as in the case of Scam, environments can be manipulated in way to provide opacity [14].

3. SELFLESS INHERITANCE

In this section, we give demonstrations on how the three aforementioned forms of inheritance can be implemented by resetting the enclosing scopes of the entities making up objects.

3.1 Extension Inheritance

In order to provide inheritance by manipulating scope, it must be possible to both get and set the static scope, at runtime, of objects and function closures. We will postulate two functions for these tasks: *getEnclosingScope* and *setEnclosingScope*. While at first glance it may seem odd to change a static scope at runtime, in environment-passing interpreters, these functions translate into getting and setting a single pointer, respectively.

Recall that in extension inheritance, the subclass strictly adds new features to a superclass and that a subclass object and a superclass object are indistinguishable, behaviour-wise, with regards to the features provided by the superclass. Delegation is clearly a simple (if not the simplest) way to provide this type of inheritance [15,16]. When a subclass object receives a message it cannot handle, it simply forwards the message to the delegate. However, roughly the same effect can be achieved via manipulation of scope. Consider two objects, *child* and *parent*. The extension inheritance of *child* from *parent* can be implemented with the following pseudocode:

```
setEnclosingScope(parent,getEnclosingScope(child));
setEnclosingScope(child,parent);
```

As a concrete example, consider the following Scam program:

```
(define (c) "happy")
(define (parent)
    (define (b) "slap")
    this)
(define (child)
    (define (a) "jacks")
    (define temp (parent))
    (setEnclosingScope temp (getEnclosingScope this))
    (setEnclosingScope this temp)
    this)

(define obj (child))
(inspect ((dot obj a)))
(inspect ((dot obj b)))
(inspect ((dot obj c)))
```

Running this program yields the following output:

```
((dot obj a)) is jacks
((dot obj b)) is slap
((dot obj c)) is happy
```

The call to *a* immediately finds the child's method. The call to *b* results in a search of the child. Failing to find a binding for *b* in *child*, the enclosing scope of *child* is searched. Since the enclosing scope of *child* has been reset to *parent*, *parent* is searched for *b* and a binding is found. In the final call to *c*, a binding is not found in either the child or the parent, so the enclosing scope of *parent* is searched. That scope has been reset to *child*'s enclosing scope. There a binding is found. So even if the parent object is created in a scope different from the child, the correct behaviour ensues.

For an arbitrarily long inheritance chain, *p1* inherits from *p2*, which inherits from *p3* and so on, the most distant ancestor of the child object receives the child's enclosing scope:

```
setEnclosingScope(pN,getEnclosingScope(p1))
setEnclosingScope(p1,p2);
setEnclosingScope(p2,p3);
...
setEnclosingScope(pN-1,pN);
```

It should be noted that the code examples in this and the next sections hard-wire the inheritance manipulations. As will be seen further on, Scam automates these tasks.

3.2 Reification Inheritance

As stated earlier, reification inheritance concerns a subclass fleshing out a partially completed implementation by the superclass. A consequence of this finishing aspect is that, unlike extension inheritance, the superclass must have access to subclass methods. A typical approach to handling this problem is rather inelegant, passing a reference to the original object as hidden parameter to all methods. Within method bodies, method calls are routed through this reference. This approach is a prime example of what Anders Hejlsberg, architect of C#, calls “simplexity”, whereby a simpler interface hides the complexity underneath. Simplexity generally makes things easier to use, if one's usage corresponds with *anticipated* uses. In novel circumstances, however, resulting behaviour can be very difficult to comprehend and/or explain [5,17].

The approach, as given in the previous section, for extension inheritance does not work for reification inheritance. Suppose a parent method references a method provided by the child. In an environment-passing interpreter, when a function definition is encountered, a closure is created and this closure holds a pointer to the definition environment. It is this pointer that implements static scoping in such interpreters.

For parent methods, then, the enclosing scope is the parent. When the function body of the method is being evaluated, the reference to the method supplied by the child goes unresolved, since it is not found in the parent method. The enclosing scope of the parent method, the parent itself, is searched next. The reference remains unresolved. Next the enclosing scope of the parent is searched, which has been reset to the enclosing scope of the child. Again, the reference goes unresolved (or resolved by happenstance should a binding appear in some enclosing scope of the child).

The solution to this problem is to reset the enclosing scopes of the parent methods to the child. In pseudocode:

```
setEnclosingScope(parent,getEnclosingScope(child));
setEnclosingScope(child,parent);
for each method m of parent:
    setEnclosingScope(m,child)
```

Now, reification inheritance works as expected. Here is an example:

```
(define (parent)
    (define (ba) (string+ (b) (a)))
    (define (b) "slap")
    this)
(define (child)
    (define (a) "jacks")
    (define temp (parent))
    (setEnclosingScope temp (getEnclosingScope this))
    (setEnclosingScope this temp)
    (setEnclosingScope (dot temp ba) this)
    this)

(define obj (child))
(inspect ((dot obj ba)))
```

In the example, the function *ba* references a function *a* that the parent does not provide, but the child does. The output of this program is:

```
((dot obj ba)) is "slapjacks"
```

As can be seen, the reference to *a* in the function *ba* is resolved correctly, due to the resetting of *ba*'s enclosing scope to *child*.

For longer inheritance chains, the pseudocode of the previous section is modified accordingly:

```
setEnclosingScope(pN,getEnclosingScope(p1))
setEnclosingScope(p1,p2);
for each method m of p2: setEnclosingScope(m,p1)
setEnclosingScope(p2,p3);
for each method m of p3: setEnclosingScope(m,p1)
...

setEnclosingScope(pN-1,pN)
for each method m of pN: setEnclosingScope(m,p1)
```

All ancestors of the child have the enclosing scopes of their methods reset.

3.3 Variation inheritance

Variation inheritance captures the idea of a subclass overriding a superclass method. If functions are naturally virtual (as in Java), then the overriding function is always called preferentially over the overridden function.

If *child* is redefined as follows:

```
(define (child)
    (define (b) "jumping")
    (define (a) "jacks")
    (define temp (parent))
    (setEnclosingScope temp (getEnclosingScope this))
    (setEnclosingScope this temp)
    (setEnclosingScope (dot temp ab) this)
    this)
```

with parent defined as before, then the new version of *b* overrides the parent version. The output now becomes:

```
((dot obj ba)) is "jumpingjacks"
```

This demonstrates that both reification and variation inheritance can be implemented using the same mechanism. Another benefit is that instance variables and instance methods are treated uniformly. Unlike virtual functions in Java and C++, instance variables in those languages shadow superclass instance variables of the same name, but only for subclass methods. For superclass methods, the superclass version of the instance variable is visible, while the subclass version is shadowed. With this approach, both instance variables and instance methods are virtual, eliminating the potential error of shadowing a superclass instance variable. Here is an example:

```
(define (parent)
    (define x 0)
    (define (toString) (string+ "x:" x))
    this)
(define (child)
    (define x 1)
    (define temp (parent))
    (setEnclosingScope temp (getEnclosingScope this))
    (setEnclosingScope this temp)
    (setEnclosingScope (dot temp toString) this)
    this)

(define p-obj (parent))
(define c-obj (child))

(inspect ((dot p-obj toString)))
(inspect ((dot c-obj toString)))
```

Note that *p-obj* points to a pure *parent* object and *c-obj* points to a child object that inherits from *parent*. The output:

```
((dot p-obj toString)) is "x:0"
((dot c-obj toString)) is "x:1"
```

demonstrates the virtuality of the instance variable *x*. Even though the program, in the last line, calls the superclass version of *toString*, the subclass version of *x* is referenced.

4. IMPLEMENTING SELFLESS INHERITANCE

Since environments are objects in Scam, implementing the *getEnclosingScope* and *setEnclosingScope* functions are trivial:

```
(define (setEnclosingScope a b) (assign (dot a __context) b))
(define (getEnclosingScope a) (dot a __context))
```

The enclosing scopes of environments and closures are stored in the field *__context*. Moreover, the task of resetting the enclosing scopes of the parties involved can be automated. Scam provides a library, written in Scam that provides a number of inheritance mechanisms. The first (and simplest) is ad hoc inheritance. Suppose we have objects *a*, *b*, and *c* and we wish to inherit from *b* and *c* (and if both *b* and *c* provide the same functionality, we prefer *b*'s implementation). To do so, we call the *mixin* function:

```
(mixin a b c)
```

A definition of *mixin* could be:

```
(define (mixin child @)  ; @ points to a list of remaining args
    (define outer (getEnclosingScope child))
    (define spot child)
    (while (not (null? (cdr @)))
        (define current (car @))
        (resetClosures current child)
        (setEnclosingScope spot current)
        (assign spot current)
        (assign @ (cdr @))
    (setEnclosingScope (car @) outer)
    (resetClosures (car @) child))
```

where *resetClosures* is tasked to set the enclosing scopes of the methods of current object to the child. Another supported type of inheritance emulates the *extends* operation in Java. For this type of inheritance, the convention is that an object must declare a parent. In the constructor for an object, the parent instance variable is set to the parent object, usually obtained via the parent constructor. Here is an example:

```
(define (b)
    (define parent nil)
    ...
    this)
(define (a)
    (define parent (b))     ;setting the parent
    ...
    this)
```

Now, to instantiate an object, the *new* function is called:

```
(define obj (new (a)))
```

The *new* function follows the parent pointers to reset the enclosing scopes appropriately. Here is a possible implementation of *new*, which follows the definition of *mixin* closely:

```
(define (new child)
    (define outer (getEnclosingScope child))
    (define spot child)
    (define current (dot spot parent))
    (while (not (null? current))
        (resetClosures current child)
        (setEnclosingScope spot current)
        (assign spot current)
```

```
        (define current (dot spot parent)))
    (setEnclosingScope spot outer)
    (resetClosures spot child))
```

The actual implementation of *new* allows the most distant ancestor to forgo a parent instance variable. Other forms of inheritance are possible as well. Thus, the flexibility of selfless inheritance does not require inheritance to be built into the language.

5. A NOTE ON DARWINIAN VS. LAMARCKIAN INHERITANCE

The behaviour of the inheritance scheme implemented in this paper differs from the inheritance schemes of the major industrial-strength languages in one important way. In Java, for example, if a superclass method references a variable defined in an outer scope (this can happen with nested classes), those references are resolved the same way, *whether or not* an object of that class was instantiated as a stand-alone object or as part of an instantiation of a subclass object. This is reminiscent of the inheritance theory of Jean-Baptiste Lamarck, who postulated that the environment influences inheritance. In Java, the superclass retains traces of its environment which can influence the behaviour of a subclass object.

With selfless inheritance, the static scopes of the superclass objects are ultimately replaced with the static scope of the subclass object, a purely Darwinian outcome. The superclass objects contribute methods and instance variables, but none of the environmental influences. Thus, the subclass object must provide bindings for the non-local references either through its own definitions or in its definition environment.

An argument can be made, in either case, as to which is the proper way to inherit. However, consider the situation where a superclass references the decidedly non-local plus operator (in Scam and Scheme, '+' is simply a variable bound to the addition function). Suppose you wish to instrument '+'; you wish to count the number of additions by overloading '+' in the environment of the subclass. Under Lamarckian inheritance, only additions performed by the subclass will be counted; additions by the superclass will not contribute to the total. Under Darwinian inheritance, additions in the subclass and superclass will both be counted.

For Scam, it is possible, when resetting the enclosing scopes of closures, to also precompile all non-local references, forcing Lamarckian inheritance.

6. CONCLUSIONS

Selfless inheritance, whereby object references are not passed to subclass and superclass methods, either explicitly or implicitly, is shown to be possible. Selflessness can be achieved by manipulating static scopes dynamically in an environment passing interpreter and is consistent with the notions of how extension, reification, and variation inheritance should behave. In terms of Reddy's [9] hierarchy of object-oriented languages, the simplest of the four languages, ObjectTalk, when augmented as above, is equivalent to the most complex of the four, SmallTalk, but without a great deal of SmallTalk's syntactic and semantic burden. Indeed, recently there has been some push back on traditional implementations of inheritance, substituting delegation to accomplish similar goals [18]. Selfless inheritance seems to combine the best of traditional inheritance and delegation.

This simple approach to inheritance semantics yields many benefits. Among them are a small, expressive, yet easily understood inheritance sub-language, both in terms of syntax and semantics. Another benefit is ease of implementation, especially in the amount of overhead in the language processor to support inheritance. A third benefit is a consistent treatment of instance variables and instance methods; since both are virtual, a single explanation of virtuality suffices to cover both. Finally, although selfless inheritance naturally implements Darwinian-style inheritance, it does not preclude Lamarckian forms.

REFERENCES

[1] H. Geffner and T. Verna, "Inheritance = chaining + defeat," in Proceedings of the Fourth International Symposium on Methodologies for Intelligent Systems (Z. Raz, ed.), (Amsterdam), pp. 411–418, North-Holland, 1989.

[2] A. Taivalsaari, "Cloning is inheritance," Computer Science Report WP-18, University of Jyvaskyla, Finland, 1991.

[3] L. A. Stein, "Delegation is inheritance," in Conference proceedings on Object-oriented programming systems, languages and applications, pp. 138–146, ACM Press, 1987.

[4] W. R. Cook, W. L. Hill, and P. S. Canning, "Inheritance is not subtyping," Information and Computation, vol. 114, no. 2, pp. 329–350, 1994.

[5] A. Taivalsaari, "On the notion of inheritance," Comp. Surveys, vol. 28, pp. 438–479, Sept. 1996.

[6] D. S. Touretzky, The mathematics of inheritance systems. PhD thesis, Carnegie-Mellon University, 1984.

[7] L. Cardelli, "A semantics of multiple inheritance," Lecture Notes in CS, no. 173, p. 51, 1984.

[8] W. R. Cook and J. Palsberg, "A denotational semantics of inheritance and correctness," Information and Computation, vol. 114, no. 2, pp. 329–350, 1994.

[9] U. S. Reddy, "Objects as closures: Abstract semantics of object oriented languages," in Proceedings of the 1988 ACM Conference on Lisp and Functional Programming, pp. 289–297, ACM, ACM, July 1988.

[10] A. Taivalsaari, "Simplifying javascript with concatenation-based prototype inheritance," tech. rep., Tampere University of Technology, 2009.

[11] B. Meyer, "The many faces of inheritance: A taxonomy of taxonomy," IEEE Computer, vol. 29, pp. 105–110, May 1996.

[12] J. C. Lusth, "The Scam Programming Language," , http://beastie.cs.ua.edu/scam/, 2011.

[13] J. C. Lusth, "Unified selection from lists, arrays, and objects," Computer Languages, Systems and Structures, vol. 28, pp. 289–305, 2002.

[14] J. C. Lusth and R. S. Bowman, "A minimalist approach to objects," Computer Languages, Systems and Structures, vol. (in press), 2004.

[15] H. Lieberman, "Using prototypical objects to implement shared behavior in object oriented systems," in OOPSLA '86 Conference Proceedings, pp. 214–223, 1986.

[16] D. Ungar and R. Smith, "Self: the power of simplicity," in OOPSLA '87 Conference Proceedings, pp. 227–242, 1987.

[17] P. H. Frohlich, "Inheritance decomposed," Position Paper, July 11 2002.

[18] H. Kegel and F. Steimann, "Systematically refactoring inheritance to delegation in java," in Proceedings of the 30th international conference on Software engineering, ICSE '08, (New York, NY, USA), pp. 431–440, ACM, 2008.